SSD-TR-91-10

AEROSPACE REPORT NO.
TR-0090(5920-07)-3





# Example Proofs Using Offline Characterizations of Procedures in the State Delta Verification System (SDVS)

Prepared by

J. V. COOK and J. E. DONER
Computer Systems Division

14 December 1990

Prepared for

SPACE SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Base
P. O. Box 92960
Los Angeles, CA 90009-2960

Engineering Group

THE AEROSPACE CORPORATION
El Segundo, California

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245-4691, under Contract No. F04701-88-C-0089 with the Space Systems Division, P. O. Box 92960, Los Angeles, CA 90009-2960. It was reviewed and approved for The Aerospace Corporation by C. A. Sunshine, Principal Director, Computer Science and Technology Subdivision. Mike Pentony, Lt, USAF, was the project officer for the Mission-Oriented Investigation and Experimentation (MOIE) program.

This report has been reviewed by the Public Affairs Office (PAS) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication. Publication of this report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.

MIKE PENTONY, Lt, USAF
MOIE Program Officer
SSD/SDEFS

JONATHAN M. EMMES, Maj, USAF
MOIE Program Manager
PL/WCO OL-AH

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3 DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>TR-0090(5920-07)-3 | 5 MONITORING ORGANIZATION REPORT NUMBER(S)<br>SSD-TR-91-10 |
| 6a NAME OF PERFORMING ORGANIZATION<br>The Aerospace Corporation<br>6b. OFFICE SYMBOL *(If applicable)* | 7a. NAME OF MONITORING ORGANIZATION<br>Space Systems Division<br>Air Force Systems Command |
| 6c ADDRESS *(City, State, and ZIP Code)*<br>P. O. Box 92957<br>Los Angeles, CA 90009-2957 | 7b. ADDRESS *(City, State, and ZIP Code)*<br>Los Angeles Air Force Base, P. O. Box 92960<br>Los Angeles, CA 90009-2960 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>8b OFFICE SYMBOL *(If applicable)* | 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>F04701-88-C-0089 |
| 8c ADDRESS *(City, State, and ZIP Code)* | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO |

11. TITLE *(Include Security Classification)*

Example Proofs Using Offline Characterizations of Procedures in the State Delta Verification System (SDVS)

12. PERSONAL AUTHOR(S)

Cook, J. V.; Doner, J. E.

| 13a TYPE OF REPORT | 13b TIME COVERED<br>FROM _____ TO _____ | 14 DATE OF REPORT *(Year, Month, Day)* | 15 PAGE COUNT |
|---|---|---|---|
| | | 14 December 1990 | 45 |

16 SUPPLEMENTARY NOTATION

| 17 COSATI CODES | | | 18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)* |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Ada, Formal methods, Ada procedures, Program verification, Correctness proofs, SDVS |
| | | | |
| | | | |

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Two examples demonstrating the offline characterization of Ada procedures are presented. By "offline characterization" we mean the characterization of the actions of a procedure independent of context. The characterization is performed within the State Delta Verification System (SDVS).

| 20 DISTRIBUTION/AVAILABILITY OF ABSTRACT | 21 ABSTRACT SECURITY CLASSIFICATION |
|---|---|
| [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS | Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b TELEPHONE *(Include Area Code)* / 22c OFFICE SYMBOL |

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

# CONTENTS

# 1 Introduction

This report presents two examples demonstrating the use of an offline characterization mechanism for Ada[1] procedures. By "offline characterization" we mean the characterization of the actions of a procedure independent of context. The characterization is performed using the State Delta Verification System (SDVS) [1]. The subset of Ada in which procedures may be written is Stage 2 Ada [2], the subset of Ada recognized by SDVS in FY90. The offline characterization feature of SDVS is described in more detail in [3].

Section 2 presents background on the offline characterization mechanism, paraphrasing some of the information presented in [3]. Section 3 gives a simple example of the offline characterization of an Ada subprogram, demonstrating the use of the characterization in a proof about an Ada program containing the subprogram. Section 4 gives a more complicated example, involving the offline characterization of two subprograms, one of which utilizes the other. Section 5 concludes this report.

[1] Ada is a registered trademark of the U. S. Government - Ada Joint Program Office.

## 2 Background

SDVS' offline characterization facility comprises the following three commands:

- the *createadalemma* command, which is used to define a lemma about an Ada procedure, in the form of a state delta, and which collects other necessary descriptive information from the user;
- the *proveadalemma* command, which sets up an environment within which the lemma (state delta) can be proved—this must be at the top level of symbolic execution, and we do not allow lemmas dependent on the dynamic program-execution context; and
- the *invokeadalemma* command, which uses a previously created lemma as a template to construct a usable state delta, which in turn is then applied to mimic the actions of the procedure call.

Perhaps the best way to discuss these commands is through examples. In sections to follow, we present annotated SDVS sessions in which Ada lemmas are created, proved, and invoked. The target programs are simple, but adequate for the purposes of testing and illustration.

At various places in the discussion of the examples, we need to refer to specific steps in the symbolic execution of procedure calls. This subject was discussed at length in [3]; for the reader's convenience, we list the steps here:

(I) Declarations of the formal parameters of the procedure are processed: The universe of places is expanded to include new places u and v.

(II) The actual parameters are evaluated, and the resulting values are bound to the places u and v.

(III) The declarations of the local variables of the procedure are processed: The universe of places is expanded to include a new place a.

(IV) The body of the procedure is executed symbolically.

(V) Undoing (III): The local variables are undeclared, so a is no longer among the places.

(VI) `in out` and `out` formal parameter values are assigned to the corresponding actual parameters: These values are determined and bound to the appropriate places.

(VII) Undoing (I): The formal parameters are undeclared, so u and v are deleted from the universe of places.

# 3 Example I: The exchange procedure

Our first example program includes a two-parameter procedure that exchanges the values of its parameters (two integers), as well as a main program to test the procedure. This program is shown below.

```
PROCEDURE xtest IS
    x, y, z :  integer := 1;
    PROCEDURE exchange(a, b :  IN OUT integer) IS
        c :  integer;
    BEGIN
        c := a;
        a := b;
        b := c;
    END exchange;
BEGIN
    get(x);
    get(y);
    get(z);
    exchange(x, y);
    exchange(y, z);
    put(x);
    put(y);
    put(z);
END xtest;
```

The lemma will simply assert, in the form of a state delta, the fact that the procedure exchanges its parameters. It will be invoked twice in the proof of a state delta describing the effect of the program as a whole, which is simply this: if the input stream consists of three integers `i`, `j`, and `k`, then the output stream will consist of the integers `j`, `k`, and `i`, in that order.

In the annotated sample session to follow, what the user types is shown in *italics*, whereas the computer's output is in `typewriter-style`.

First, we read a file that contains the predefined state delta describing the action of the test program.

```
<sdvs.1> read
   path name[foo.proofs]: testproofs/xtest.proofs

 Definitions read from file "testproofs/xtest.proofs"
   -- (xtest.sd,xtest.proof)

<sdvs.2> pp
   object: xtest.sd

 [sd pre: (ada(xtest.ada))
   comod: (all)
     mod: (all)
```

```
post: (#xtest\stdout[1] = .xtest\stdin[2],
       #xtest\stdout[2] = .xtest\stdin[3],
       #xtest\stdout[3] = .xtest\stdin[1])]
```

Next, we use the *adatr* command to parse and translate the program file.

```
<sdvs.2> adatr
   path name[foo.ada]: testproofs/xtest.ada

 Parsing Stage 2 Ada file -- "testproofs/xtest.ada"

 Translating Stage 2 Ada file -- "testproofs/xtest.ada"
```

The *createadalemma* command is used to create the lemma, which will be a certain state delta.

```
<sdvs.3> createadalemma
        lemma name: exchange.lemma
         file name: testproofs/xtest.ada
   subprogram name: exchange
    qualified name: xtest.exchange
   preconditions[]:
        mod list[]: a,b
    postconditions: #a=.b,#b=.a

 createadalemma -- [sd pre: (.xtest\pc = at(xtest.exchange))
                    comod: (all)
                      mod: (xtest\pc,a,b)
                     post: (#a = .b,#b = .a,
                            #xtest\pc
                               = exited(xtest.exchange))]
```

Notice that the system supplies additional entries for the state delta besides those given by the user. The condition

$$.\texttt{xtest\pc} = \texttt{at(xtest.exchange)}$$

becomes true exactly when the symbolic execution of a call to procedure exchange has completed Step II. Similarly, the condition

$$\#\texttt{xtest\pc} = \texttt{exited(xtest.exchange)}$$

will be true when the symbolic execution of a call has completed Step V. Also, xtest\pc should always be part of the mod list for a state delta dealing with any part of the program xtest. To identify fully the code to which the lemma refers, one must supply a full path name to the file, and a *fully qualified* procedure name. The fully qualified name in this case is "xtest.exchange;" in general, it is a complete list of procedure and block names, in the order of containment and ending with the given procedure, with all names separated by periods. (If a containing block is unnamed, the parser supplies an internal name, which in principle could be used in this context; however, it is recommended that the user explicitly name the containing block.)

The *provcadalemma* command causes SDVS to set up the environment for proving the lemma.

```
<sdvs.4> proveadalemma
    Ada lemma name: exchange.lemma
           proof[]:
```

A name of a previously created and saved proof may be entered at this point, or, as in this case, an empty line signals an interactive proof.

The *proveadalemma* command must be issued at SDVS' top level, that is, in an environment in which no variable bindings have occurred. The system automatically emits and applies state deltas to set up the environment appropriate for proving the lemma:

```
open -- [sd pre: (alldisjoint(xtest,.xtest),
                  covering(.xtest,xtest\pc,exception,
                           xtest\stdin,xtest\stdin\ctr,
                           xtest\stdout,xtest\stdout\ctr,x,
                           y,z),
                  <adatr exchange (a, ...);>)
          comod: (all)
            mod: (all)
           post. ([sd pre: (.xtest\pc = at(xtest.exchange))
                     comod: (all)
                       mod: (diff(all,
                                  diff(union(xtest\pc,exception,
                                             xtest\stdin,
                                             xtest\stdin\ctr,
                                             xtest\stdout,
                                             xtest\stdout\ctr,x,y,z,
                                             a,b),
                                       union(xtest\pc,
                                             xtest\pc,a,b))))
                      post: (#a = .b,#b = .a,
                             #xtest\pc
                                = exited(xtest.exchange))])]

instantiate in q(1) -- all top-level existential variables

apply -- [sd pre: (true)
            comod: (all)
              mod: (xtest\pc,xtest)
             post: (alldisjoint(xtest,.xtest,a,b),
                    covering(#xtest,.xtest,a,b),
                    declare(a,type(integer)),declare(b,type(integer)),
                    <adatr null;>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (xtest\pc,a,b)
             post: (#a = .a,#b = .b,
                    <adatr null;>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (xtest\pc)
             post: (#xtest\pc = at(xtest.exchange),
                    <adatr null;>)]
```

```
go -- breakpoint reached

open -- [sd pre: (.xtest\pc = at(xtest.exchange))
          comod: (all)
            mod: (diff(all,
                       diff(union(xtest\pc,exception,
                                  xtest\stdin,
                                  xtest\stdin\ctr,
                                  xtest\stdout,
                                  xtest\stdout\ctr,x,y,z,a,b),
                            union(xtest\pc,xtest\pc,a,b))))
           post: (#a = .b,#b = .a,
                  #xtest\pc = exited(xtest.exchange))]
```

The environment at this point is identical to what would exist after the completion of Steps I–II of the symbolic execution of a call to the procedure. Examining the output above, we see that this environment was created by opening the proof of a state delta having (1) a precondition establishing the necessary environment, and (2) a postcondition consisting of the state delta of the lemma. The last step above opens the proof of the latter state delta. The system's response to each intermediate **apply** command (these are internally generated) shows the state delta being applied, and the **adatr** fields show the particular Ada program statement with which the currently applied state delta is associated.

The reader will notice that the last state delta opened for proof is not exactly the same as that of the lemma: the mod list is apparently more complex. This is done to allow for modification, during the proof, of new places created by declarations arising during the symbolic execution of the procedure body. An evaluation of the expression for the mod list will show that in the current context it describes no more than the places named in the original mod list. However, the value of this expression will change appropriately as other places are created through declaration, or deleted by undeclaration.

The *usable* command will help us ascertain the current state of symbolic execution.

```
<sdvs.4.5.1> usable

 No usable state deltas.


 q(1) exists c ([sd pre: (true)
                 comod: (all)
                   mod: (xtest\pc,xtest)
                  post: (alldisjoint(xtest,.xtest,c),
                         covering(#xtest,.xtest,c),
                         declare(c,type(integer)),
                         <adatr c : integer>)])
```

This shows that symbolic execution is just at the point of the declaration of the local variable in the exchange procedure—i.e., just before Step III of processing a procedure call. The next step will be an instantiation of the quantified statement **true** at this point.

```
<sdvs.4.5.1> instantiate
```

```
      existential formula: q
                   number: 1
   existential variable[]:

    instantiate in q(1) -- all top-level existential variables

<sdvs.4.5.2> usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (xtest\pc,xtest)
        post: (alldisjoint(xtest,.xtest,c),
               covering(#xtest,.xtest,c),
               declare(c,type(integer)),
               <adatr c : integer>)]


 q(1) exists c ([sd pre: (true)
                 comod: (all)
                   mod: (xtest\pc,xtest)
                  post: (alldisjoint(xtest,.xtest,c),
                         covering(#xtest,.xtest,c),
                         declare(c,type(integer)),
                         <adatr c : integer>)])

<sdvs.4.5.2> apply
   sd/number[highest applicable/once]:

     apply -- [sd pre: (true)
               comod: (all)
                 mod: (xtest\pc,xtest)
                post: (alldisjoint(xtest,.xtest,c),
                       covering(#xtest,.xtest,c),
                       declare(c,type(integer)),
                       <adatr c : integer>)]

<sdvs.4.5.3> usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (xtest\pc,c)
        post: (#c = .a,
               <adatr c := a;>)]


 No usable quantifiers.
```

The instantiation of the quantifier and the application of state deltas to effect the necessary declarations brings us to the first executable statement in the body of the procedure.

```
<sdvs.4.5.3> go
   until[]: #xtest\pc = exited(xtest.exchange)

     apply -- [sd pre: (true)
               comod: (all)
                 mod: (xtest\pc,c)
```

```
                post: (#c = .a,
                       <adatr c := a;>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (xtest\pc,a)
               post: (#a = .b,
                       <adatr a := b;>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (xtest\pc,b)
               post: (#b = .c,
                       <adatr b := c;>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (xtest\pc,xtest,c)
               post: (covering(.xtest,#xtest,c),
                       undeclare(c),
                       <adatr c : integer>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (xtest\pc)
               post: (#xtest\pc = exited(xtest.exchange),
                       <adatr null;>)]

  close -- 7 steps/applications

close -- 5 steps/applications

proveadalemma -- [sd pre: (.xtest\pc = at(xtest.exchange))
                  comod: (all)
                    mod: (xtest\pc,a,b)
                   post: (#a = .b,#b = .a,
                          #xtest\pc = exited(xtest.exchange))]
```

The facts to be proven here are sufficiently simple that the proof of the lemma closes automatically.

The lemma being proved, the next step is to reinitialize SDVS, and prove the overall state delta `xtest.sd`.

```
<sdvs.4> init
   proof name[]:

State Delta Verification System, Version 9

Restricted to authorized users only.

<sdvs.1> prove xtest.sd
   state delta:        proof[]:

 open -- [sd pre: (ada(xtest.ada))
          comod: (all)
```

```
            mod: (all)
           post: (#xtest\stdout[1] = .xtest\stdin[2],
                  #xtest\stdout[2] = .xtest\stdin[3],
                  #xtest\stdout[3] = .xtest\stdin[1])]

Complete the proof.

<sdvs.1.1> usable

u(1) [sd pre: (true)
      comod: (all)
        mod: (xtest\pc)
       post: (<adatr procedure xtest is
                    x, ... : integer := 1
                    ...
                  begin
                    get (x);
                    ...
                  end xtest;>)]


No usable quantifiers.
```

The *go* command can be used to cause the system to apply state deltas and perform instantiations until a specified condition holds.

```
<sdvs.1.1> go
   until[]: #xtest\pc = at(xtest.exchange)
```

This *go* command will bring us into the first call to the exchange procedure, in a position to apply the lemma.

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc)
           post: (<adatr procedure xtest is
                        x, ... : integer := 1
                        ...
                      begin
                        get (x);
                        ...
                      end xtest;>)]

instantiate in q(1) -- all top-level existential variables

apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc,xtest)
           post: (alldisjoint(xtest,.xtest,x,y,z),
                  covering(#xtest,.xtest,x,y,z),
                  declare(x,type(integer)),declare(y,type(integer)),
                  declare(z,type(integer)),
                  <adatr x, ... : integer := 1>)]
```

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc,x,y,z)
           post: (#x = 1,#y = 1,#z = 1,
                  <adatr x, ... : integer := 1>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc,x,xtest\stdin\ctr)
           post: (#x = .xtest\stdin[.xtest\stdin\ctr],
                  #xtest\stdin\ctr
                     = .xtest\stdin\ctr + 1,
                  <adatr get (x);>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc,y,xtest\stdin\ctr)
           post: (#y = .xtest\stdin[.xtest\stdin\ctr],
                  #xtest\stdin\ctr
                     = .xtest\stdin\ctr + 1,
                  <adatr get (y);>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc,z,xtest\stdin\ctr)
           post: (#z = .xtest\stdin[.xtest\stdin\ctr],
                  #xtest\stdin\ctr
                     = .xtest\stdin\ctr + 1,
                  <adatr get (z);>)]

instantiate in q(1) -- all top-level existential variables

apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc,xtest)
           post: (alldisjoint(xtest,.xtest,a,b),
                  covering(#xtest,.xtest,a,b),
                  declare(a,type(integer)),declare(b,type(integer)),
                  <adatr exchange (x, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc,a,b)
           post: (#a = .x,#b = .y,
                  <adatr exchange (x, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (xtest\pc)
           post: (#xtest\pc = at(xtest.exchange),
                  <adatr exchange (x, ...)>)]

go -- breakpoint reached
```

<sdvs.1.12> *usable*

```
No usable state deltas.
```

```
q(1) exists c ([sd pre: (true)
                comod: (all)
                  mod: (xtest\pc,xtest)
                 post: (alldisjoint(xtest,.xtest,c),
                        covering(#xtest,.xtest,c),
                        declare(c,type(integer)),
                        <adatr c : integer>)])
```

Symbolic execution is now at precisely the point where Steps I and II of the first call to the exchange procedure have been completed, where the next step would be the instantiation for the declaration of the local variable. Instead, we can invoke the lemma to bypass the symbolic execution of the procedure body.

```
<sdvs.1.12> invokeadalemma
   Ada lemma name: exchange.lemma

   invokeadalemma -- [sd pre: (.xtest\pc = at(xtest.exchange))
                      comod: (all)
                        mod: (xtest\pc,a,b)
                       post: (#a = .b,#b = .a,
                              #xtest\pc
                                 = exited(xtest.exchange),
                              <adatr return;>)]

<sdvs.1.13> usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (xtest\pc)
        post: (#xtest\pc = exited(xtest.exchange),
               <adatr exchange (x, ...)>)]


 No usable quantifiers.

<sdvs.1.13> apply
   sd/number[highest applicable/once]:

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc)
              post: (#xtest\pc = exited(xtest.exchange),
                     <adatr exchange (x, ...)>)]

<sdvs.1.14> usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (xtest\pc,x,y)
        post: (#x = .a,#y = .b,
               <adatr exchange (x, ...)>)]
```

```
No usable quantifiers.
```

This point is immediately following completion of Step V. Two more state deltas are applied to complete Steps VI and VII.

```
<sdvs.1.14> apply 2
   sd/number[highest applicable/once]:
   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,x,y)
              post: (#x = .a,#y = .b,
                     <adatr exchange (x, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,xtest,a,b)
              post: (covering(.xtest,#xtest,a,b),
                     undeclare(a,b),
                     <adatr exchange (x, ...)>)]

<sdvs.1.16> usable

 No usable state deltas.


 q(1) exists a!2 exists b!2 ([sd pre: (true)
                              comod: (all)
                                mod: (xtest\pc,xtest)
                               post: (alldisjoint(xtest,
                                                  .xtest,a!2,b!2),
                                      covering(#xtest,
                                               .xtest,a!2,b!2),
                                      declare(a!2,type(integer)),
                                      declare(b!2,type(integer)),
                                      <adatr exchange (y, ...)>)])
```

This is the beginning of the next Ada statement.

We go on to the point where the lemma can be invoked again, we invoke it, and we then apply the state deltas to complete the return from the call.

```
<sdvs.1.16> go
   until[]: #xtest\pc = at(xtest.exchange)

   instantiate in q(1) -- all top-level existential variables

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,xtest)
              post: (alldisjoint(xtest,.xtest,a!2,b!2),
                     covering(#xtest,.xtest,a!2,b!2),
                     declare(a!2,type(integer)),
                     declare(b!2,type(integer)),
```

```
                    <adatr exchange (y, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,a!2,b!2)
              post: (#a!2 = .y,#b!2 = .z,
                     <adatr exchange (y, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc)
              post: (#xtest\pc = at(xtest.exchange),
                     <adatr exchange (y, ...)>)]

   go -- breakpoint reached
```

<sdvs.1.20> *invokeadalemma*

Ada lemma name: *exchange.lemma*

```
   invokeadalemma -- [sd pre: (.xtest\pc = at(xtest.exchange))
                      comod: (all)
                        mod: (xtest\pc,a!2,b!2)
                       post: (#a!2 = .b!2,#b!2 = .a!2,
                              #xtest\pc
                                 = exited(xtest.exchange),
                              <adatr return;>)]
```

<sdvs.1.21> *apply 3*

```
   sd/number[highest applicable/once]:
   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc)
              post: (#xtest\pc = exited(xtest.exchange),
                     <adatr exchange (y, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,y,z)
              post: (#y = .a!2,#z = .b!2,
                     <adatr exchange (y, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,xtest,a!2,b!2)
              post: (covering(.xtest,#xtest,a!2,b!2),
                     undeclare(a!2,b!2),
                     <adatr exchange (y, ...)>)]
```

<sdvs.1.24> *usable*

```
 u(1) [sd pre: (true)
       comod: (all)
         mod: (xtest\pc,
               xtest\stdout[.xtest\stdout\ctr],
               xtest\stdout\ctr)
        post: (#xtest\stdout[.xtest\stdout\ctr] = .x,
               #xtest\stdout\ctr = .xtest\stdout\ctr + 1,
```

```
                <adatr put (x);>)]

No usable quantifiers.
```

We now simply go on through the rest of the test program.

```
<sdvs.1.12> invokeadalemma
<sdvs.1.24> go
   until[]: terminated(xtest)

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,
                     xtest\stdout[.xtest\stdout\ctr],
                     xtest\stdout\ctr)
              post: (#xtest\stdout[.xtest\stdout\ctr] = .x,
                     #xtest\stdout\ctr
                        = .xtest\stdout\ctr + 1,
                     <adatr put (x);>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,
                     xtest\stdout[.xtest\stdout\ctr],
                     xtest\stdout\ctr)
              post: (#xtest\stdout[.xtest\stdout\ctr] = .y,
                     #xtest\stdout\ctr
                        = .xtest\stdout\ctr + 1,
                     <adatr put (y);>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (xtest\pc,
                     xtest\stdout[.xtest\stdout\ctr],
                     xtest\stdout\ctr)
              post: (#xtest\stdout[.xtest\stdout\ctr] = .z,
                     #xtest\stdout\ctr
                        = .xtest\stdout\ctr + 1,
                     <adatr put (z);>)]

  close -- 26 steps/applications

<sdvs.2> ps

  << initial state >>
  proved xtest.sd <1>
  --> you are here <--

<sdvs.2>
```

The postcondition of `xtest.sd` is sufficiently simple that the system can verify it without assistance, and the proof closes automatically.

The following is the saved version of the proof presented above:

```
; -*- Syntax: Common-lisp; Package: USER; Mode: LISP -*-~%

(defsd xtest.sd
   "[sd pre: (ada(xtest.ada))
      comod: (all)
        mod: (all)
       post: (#xtest\\stdout[1]=.xtest\\stdin[2],
              #xtest\\stdout[2]=.xtest\\stdin[3],
              #xtest\\stdout[3]=.xtest\\stdin[1])]")

(defproof xtest.proof
   "(adatr \"testproofs/xtest.ada\",
     createadalemma exchange.lemma
                 file: \"testproofs/xtest.ada\"
            procedure: exchange
      qualified name: xtest.exchange
        precondition:
            mod list: (a,b)
       postcondition: (#a = .b,#b = .a),
     proveadalemma exchange.lemma
       proof:
 (go #xtest\\pc = exited(xtest.exchange),
       close),
     prove xtest.sd
       proof:
 (go #xtest\\pc = at(xtest.exchange),
  invokeadalemma exchange.lemma,
  go #xtest\\pc = at(xtest.exchange),
  invokeadalemma exchange.lemma,
  go terminated(xtest),
  close))")
```

The reader may well wonder why the Ada lemma can be invoked only after a call to the procedure has been partly processed (manually), and why afterwards we still have to apply two more state deltas to complete the call. Why should not the system be programmed to perform these instantiations and state delta applications automatically? In fact, there is no reason why this would not have worked in our example. But here, all the conditions to be proven were simple enough that they could be verified by SDVS' simplifier and propagated automatically. With more complex conditions, perhaps involving quantifiers, this would not be the case, and the user would need to assist the system in propagating these conditions through the steps at the beginning and end of the procedure call.

# 4 Example II: Finding a Maximum

Our second example is a program to find the maximum element in an integer array. It uses a subroutine for finding the maximum of two elements. Although a trivial subroutine, this will nevertheless serve to illustrate many aspects of the offline characterization, not to mention honing our skills at giving proofs of facts involving quantifiers. The target program is

```
procedure findmaxtest is
    size :  integer;
    i    :  integer;
begin
    get(size);
    inner :
        declare
            x :  array(1 ..  size) of integer;
            procedure max(i :  in integer; n :  in out integer) is
            begin
                if x(i) > n then
                    n := x(i);
                end if;
            end max;
            procedure findmax(m :  in out integer) is
                i :  integer;
            begin
                m := x(1);
                i := 2;
                while i <= size loop
                    max(i, m);
                    i := i + 1;
                end loop;
            end findmax;
        begin
            i := 1;
            while i <= size loop
                get(x(i));
                i := i + 1;
            end loop;
            findmax(i);
        end inner;
    put(i);
end findmaxtest;
```

In this example we avoid the previous procedure of stating and proving a state delta about the entire program. Instead, we state and prove a state delta, as an Ada lemma, about the `findmax` procedure. Since `findmax` itself calls another procedure, this will provide ample opportunity to see the offline characterization feature in action. In this way, we avoid the

routine and tiresome task of proving facts about the input and output streams. The overall strategy is first to prove a lemma about the `max` procedure, which will include some details needed for the later application. Then we set up and prove a lemma about `findmax` itself. This proof involves an induction, and within the step case of the induction, we apply the previously proved lemma about `max`.

```
<sdvs.1> adatr
   path name[foo.ada]:   tcstproofs/findmax.a

 Reading parse tree file for Stage 2 Ada file -- "findmax.a"

 Translating Stage 2 Ada file -- "testproofs/findmax.a"

<sdvs.2> createadalemma
        lemma name:  max.lemma
         file name:  testproofs/findmax.a
   subprogram name:  max
    qualified name:  findmaxtest.inner.max
   preconditions[]:  1 le .max.i,.max.i le .size,
                     exists j ((1 le j & j le .size) & .x[j] = .n)
        mod list[]:  n
    postconditions:  #n = .n or #n = .x[.max.i],#n ge .n,#n ge .x[.max.i],
                     exists j ((1 le j & j le .size) & .x[j] = #n)

 createadalemma -- [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.max),
                             1 le .max.i,.max.i le .size,
                             exists j ((1 le j & j le .size) & .x[j] = .n))
                    comod: (all)
                      mod: (findmaxtest\pc,n)
                     post: (#n = .n or #n = .x[.max.i],#n ge .n,
                             #n ge .x[.max.i],
                             exists j ((1 le j & j le .size) & .x[j] = #n),
                             #findmaxtest\pc = exited(findmaxtest.inner.max))]
```

The symbol `i` is declared in several places in `findmaxtest`, with inner declarations shadowing outer ones. SDVS, however, needs unique names. Therefore, the translator inserts qualifications to distinguish the various places named "`i`," and so we have `max.i` here and later `findmax.i`. Having created the lemma, our next step is to prove it.

```
<sdvs.3> proveadalemma
    Ada lemma name:  max.lemma
           proof[]:

 open -- [sd pre: (alldisjoint(findmaxtest,.findmaxtest),
                   covering(.findmaxtest,findmaxtest\pc,findmaxtest\stdin,
                            findmaxtest\stdin\ctr,findmaxtest\stdout,
                            findmaxtest\stdout\ctr,size,i,x),
                   declare(x,
                           type(array,origin(x),range(x),type(integer))),
                   declare(i,type(integer)),declare(size,type(integer)),
                   <adatr max (max.i, ...);>)
          comod: (all)
            mod: (all)
```

```
        post: ([sd pre: (.findmaxtest\pc = at(findmaxtest.inner.max),
                         1 le .max.i,.max.i le .size,
                         exists k ((1 le k & k le .size) &
                                   .x[k] = .n))
                comod: (all)
                  mod: (diff(all,
                             diff(union(findmaxtest\pc,
                                        findmaxtest\stdin,
                                        findmaxtest\stdin\ctr,
                                        findmaxtest\stdout,
                                        findmaxtest\stdout\ctr,size,i,
                                        x,max.i,n),.
                                  union(findmaxtest\pc,n))))
                 post: (#n = .n or #n = .x[.max.i],#n ge .n,
                        #n ge .x[.max.i],
                        exists k ((1 le k & k le .size) &
                                  .x[k] = #n),
                        #findmaxtest\pc = exited(findmaxtest.inner.max))])]

instantiate in q(1) -- all top-level existential variables

apply -- [sd pre: (true)
          comod: (all)
            mod: (findmaxtest\pc,findmaxtest)
           post: (alldisjoint(findmaxtest,.findmaxtest,max.i,n),
                  covering(#findmaxtest,.findmaxtest,max.i,n),
                  declare(max.i,type(integer)),
                  declare(n,type(integer)),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (findmaxtest\pc,max.i,n)
           post: (#max.i = .max.i,#n = .n,
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (findmaxtest\pc)
           post: (#findmaxtest\pc = at(findmaxtest.inner.max),
                  <adatr null;>)]

go -- breakpoint reached

open -- [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.max),
                  1 le .max.i,.max.i le .size,
                  exists k ((1 le k & k le .size) & .x[k] = .n))
           comod: (all)
             mod: (diff(all,
                        diff(union(findmaxtest\pc,findmaxtest\stdin,
                                   findmaxtest\stdin\ctr,
                                   findmaxtest\stdout,
                                   findmaxtest\stdout\ctr,size,i,x,max.i,
                                   n),
                             union(findmaxtest\pc,n))))
            post: (#n = .n or #n = .x[.max.i],#n ge .n,
                   #n ge .x[.max.i],
```

```
                    exists k ((1 le k & k le .size) & .x[k] = #n),
                    #findmaxtest\pc = exited(findmaxtest.inner.max))]

<sdvs.3.5.1>  usable

 u(1) [sd pre: (~(.x[.max.i] gt .n),.max.i ge 1,.max.i le .size)
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr not (x (i) > n)>)]

 u(2) [sd pre: (.x[.max.i] gt .n,.max.i ge 1,.max.i le .size)
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr if x (i) > n
                        then n := x (i);
                    end if;>)]


 q(1) exists k ((1 le k & k le size\18) & .x[k] = n\13)
```

The system automatically opens a state delta and applies state deltas to create the environment suitable for proving `max.lemma`. The last step was to open the `max.lemma` state delta. Since the body of `max` begins with an `if` statement, the usable state deltas have preconditions for two possibilities, and we must break the proof into two cases.

```
 <sdvs.3.5.1>  cases
   case predicate:  .x[.max.i] gt .n

     cases -- .x[.max.i] gt .n

       open -- [sd pre: (.x[.max.i] gt .n)
                 comod: (all)
                   mod: (diff(all,
                              diff(union(findmaxtest\pc,findmaxtest\stdin,
                                         findmaxtest\stdin\ctr,
                                         findmaxtest\stdout,
                                         findmaxtest\stdout\ctr,size,i,x,
                                         max.i,n),
                                   union(findmaxtest\pc,n))))
                  post: (#n = n\13 or #n = x\19,#n ge n\13,#n ge x\19,
                         exists k ((1 le k & k le size\18) &
                                   .x[k] = #n),
                         #findmaxtest\pc = exited(findmaxtest.inner.max))]

<sdvs.3.5.1.1.1>  go
   until[]:  #findmaxtest\pc = exited(findmaxtest.inner.max)

         apply -- [sd pre: (.x[.max.i] gt .n,.max.i ge 1,.max.i le .size)
                   comod: (all)
                     mod: (findmaxtest\pc)
                    post: (<adatr if x (i) > n
                                    then n := x (i);
                                end if;>)]

         apply -- [sd pre: (.max.i ge 1,.max.i le .size)
```

```
                comod: (all)
                  mod: (findmaxtest\pc,n)
                 post: (#n = .x[.max.i],
                        <adatr n := x (i);>)]

        apply -- [sd pre: (true)
                comod: (all)
                  mod: (findmaxtest\pc)
                 post: (#findmaxtest\pc = exited(findmaxtest.inner.max),
                        <adatr null;>)]

        go -- breakpoint reached
```

The *cases* command set up the environment for the body of the `if` statement; then the *go* command advanced symbolic execution to the end of the `max` procedure. Now we must see to the postcondition to be established.

```
<sdvs.3.5.1.1.4> whynotgoal
   simplify?[no]:

 g(4) exists k ((1 le k & k le size\18) & .x[k] = #n)

<sdvs.3.5.1.1.4> usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (findmaxtest\pc,n)
        post: (#n = .n,
               <adatr null;>)]


 q(1) exists k ((1 le k & k le size\18) & .x[k] = n\13)

<sdvs.3.5.1.1.4> ppeq
   expression: .n
 eqclass = x\19

<sdvs.3.5.1.1.4> simp
   expression: n\13 = .n

 false
```

The symbol `n\13` here refers to the original value of `n`, not the current one. To prove the goal in this case, we note that `.max.i` is a value for `k` with the following required property:

```
<sdvs.3.5.1.1.4> instantiate
      existential formula: g
                   number: 4
   existential variable[]: k
         instantiated by: .max.i
   existential variable[]:

        instantiate in goal 4 -- .max.i for k.
```

```
      close -- 4 steps/applications

      open -- [sd pre: (~(.x[.max.i] gt .n))
                comod: (all)
                  mod: (diff(all,
                             diff(union(findmaxtest\pc,findmaxtest\stdin,
                                        findmaxtest\stdin\ctr,
                                        findmaxtest\stdout,
                                        findmaxtest\stdout\ctr,size,i,x,
                                        max.i,n),
                                  union(findmaxtest\pc,n))))
                 post: (#n = n\13 or #n = x\19,#n ge n\13,#n ge x\19,
                        exists k ((1 le k & k le size\18) &
                                  .x[k] = #n),
                        #findmaxtest\pc = exited(findmaxtest.inner.max))]

  Complete the proof.
```

The system opens the other case. No statements are executed in this branch of the `if`, so the proof of the `max.lemma` will close automatically.

```
<sdvs.3.5.1.2.1> go
   until[]: #findmaxtest\pc = exited(findmaxtest.inner.max)

        apply -- [sd pre: (~(.x[.max.i] gt .n),.max.i ge 1,
                           .max.i le .size)
                  comod: (all)
                    mod: (findmaxtest\pc)
                   post: (<adatr not (x (i) > n)>)]

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (findmaxtest\pc)
                   post: (#findmaxtest\pc = exited(findmaxtest.inner.max),
                          <adatr null;>)]

       close -- 2 steps/applications

      join -- [sd pre: (true)
               comod: (all)
                 mod: (diff(all,
                            diff(union(findmaxtest\pc,findmaxtest\stdin,
                                       findmaxtest\stdin\ctr,
                                       findmaxtest\stdout,
                                       findmaxtest\stdout\ctr,size,i,x,
                                       max.i,n),
                                 union(findmaxtest\pc,n))))
                post: (#n = n\13 or #n = x\19,#n ge n\13,#n ge x\19,
                       exists k ((1 le k & k le size\18) & .x[k] = #n),
                       #findmaxtest\pc = exited(findmaxtest.inner.max))]

     close -- 1 steps/applications

   close -- 5 steps/applications

   proveadalemma -- [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.max),
```

```
                1 le .max.i,.max.i le .size,
                exists k ((1 le k & k le .size) &
                          .x[k] = .n))
          comod: (all)
            mod: (findmaxtest\pc,n)
           post: (#n = .n or #n = .x[.max.i],#n ge .n,
                #n ge .x[.max.i],
                exists k ((1 le k & k le .size) &
                          .x[k] = #n),
                #findmaxtest\pc = exited(findmaxtest.inner.max))]
```

Now we create a lemma for the **findmax** procedure and initiate its proof.

```
<sdvs.3> createadalemma
        lemma name: findmax.lemma
         file name: testproofs/findmax.a
   subprogram name: findmax
    qualified name: findmaxtest.inner.findmax
   preconditions[]: 1 le .size
        mod list[]: m
    postconditions: forall j (1 le j & j le .size -> .x[j] le #m),
                    exists j ((1 le j & j le .size) & #m = .x[j])

 createadalemma -- [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.findmax),
                             1 le .size)
                    comod: (all)
                      mod: (findmaxtest\pc,m)
                     post: (forall j (1 le j & j le .size --> .x[j] le #m),
                            exists j ((1 le j & j le .size) &
                                      #m = .x[j]),
                            #findmaxtest\pc
                               = exited(findmaxtest.inner.findmax))]

<sdvs.4> proveadalemma
    Ada lemma name: findmax.lemma
             proof[]:

 open -- [sd pre: (alldisjoint(findmaxtest,.findmaxtest),
                   covering(.findmaxtest,findmaxtest\pc,findmaxtest\stdin,
                            findmaxtest\stdin\ctr,findmaxtest\stdout,
                            findmaxtest\stdout\ctr,size,i,x),
                   declare(x,
                           type(array,origin(x),range(x),type(integer))),
                   declare(i,type(integer)),declare(size,type(integer)),
                   <adatr findmax (m);>)
             comod: (all)
               mod: (all)
              post: ([sd pre: (.findmaxtest\pc = at(findmaxtest.inner.findmax),
                               1 le .size)
                        comod: (all)
                          mod: (diff(all,
                                     diff(union(findmaxtest\pc,
                                                findmaxtest\stdin,
                                                findmaxtest\stdin\ctr,
                                                findmaxtest\stdout,
                                                findmaxtest\stdout\ctr,size,i,
```

```
                                        x,m),
                                    union(findmaxtest\pc,m))))
                        post: (forall j (1 le j & j le .size --> .x[j] le #m),
                               exists j ((1 le j & j le .size) &
                                         #m = .x[j]),
                               #findmaxtest\pc
                                  = exited(findmaxtest.inner.findmax))])]

  instantiate in q(1) -- all top-level existential variables

  apply -- [sd pre: (true)
            comod: (all)
              mod: (findmaxtest\pc,findmaxtest)
             post: (alldisjoint(findmaxtest,.findmaxtest,m),
                    covering(#findmaxtest,.findmaxtest,m),
                    declare(m,type(integer)),
                    <adatr null;>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (findmaxtest\pc,m)
             post: (#m = .m,
                    <adatr null;>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (findmaxtest\pc)
             post: (#findmaxtest\pc = at(findmaxtest.inner.findmax),
                    <adatr null;>)]

  go -- breakpoint reached

  open -- [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.findmax),
                    1 le .size)
            comod: (all)
              mod: (diff(all,
                         diff(union(findmaxtest\pc,findmaxtest\stdin,
                                    findmaxtest\stdin\ctr,
                                    findmaxtest\stdout,
                                    findmaxtest\stdout\ctr,size,i,x,m),
                              union(findmaxtest\pc,m))))
             post: (forall j (1 le j & j le .size --> .x[j] le #m),
                    exists j ((1 le j & j le .size) & #m = .x[j]),
                    #findmaxtest\pc = exited(findmaxtest.inner.findmax))]

<sdvs.4.5.1> usable

No usable state deltas.


q(1) exists findmax.i ([sd pre: (true)
                        comod: (all)
                          mod: (findmaxtest\pc,findmaxtest)
                         post: (alldisjoint(findmaxtest,.findmaxtest,
                                            findmax.i),
                                covering(#findmaxtest,.findmaxtest,
                                         findmax.i),
```

```
                                declare(findmax.i,type(integer)),
                                <adatr i : integer>)])

<sdvs.4.5.1> applydecls

    instantiate in q(1) -- all top-level existential variables

    apply -- [sd pre: (true)
              comod: (all)
                mod: (findmaxtest\pc,findmaxtest)
               post: (alldisjoint(findmaxtest,.findmaxtest,findmax.i),
                      covering(#findmaxtest,.findmaxtest,findmax.i),
                      declare(findmax.i,type(integer)),
                      <adatr i : integer>)]

    applydecls -- declaration elaboration complete.

<sdvs.4.5.3> usable

 u(1) [sd pre: (1 ge 1,1 le .size)
       comod: (all)
         mod: (findmaxtest\pc,m)
        post: (#m = .x[1],
               <adatr m := x (1);>)]


 No usable quantifiers.

<sdvs.4.5.3> go
   until[]: .findmax.i = 2

    apply -- [sd pre: (1 ge 1,1 le .size)
              comod: (all)
                mod: (findmaxtest\pc,m)
               post: (#m = .x[1],
                      <adatr m := x (1);>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (findmaxtest\pc,findmax.i)
               post: (#findmax.i = 2,
                      <adatr i := 2;>)]

    go -- no more declarations or statements

 <sdvs.4.5.5> usable

  u(1) [sd pre: (~(.findmax.i le .size))
        comod: (all)
          mod: (findmaxtest\pc)
         post: (<adatr while i <= size
                       loop
                         max (i, ...);
                         ...
                       end loop;>)]

  u(2) [sd pre: (.findmax.i le .size)
```

```
      comod: (all)
        mod: (findmaxtest\pc)
       post: (<adatr while i <= size
                     loop
                       max (i, ...);
                       ...
                     end loop;>)]

No usable quantifiers.
```

Symbolic execution is now at the entry to the **while** loop in `findmax`. We prove the desired postcondition of this loop by induction on `findmax.i` from 2 to `.size+1`. It is necessary to provide expressions for the comod and mod lists of the induction state delta; these lists, even when evaluated in a universe temporarily expanded by inner declarations, still specify the constancy of places in the present universe, other than the ones listed, and also permit the modification of the ones listed and any new places created internally to the loop. Calling a procedure within the loop involves just such a temporary expansion of the universe. We assign the current value of the universe of places to a variable `universe1`, and then an expression of form `diff(universe1, . . .)` can define the set of places that do not change, even when the expression is evaluated in a universe larger than `universe1`. Similarly, an expression of the form `diff(all,universe1)` captures all the places that have been created between the present time and the time the expression is evaluated. We also name the two currently usable state deltas:

```
<sdvs.4.5.5> let
   new variable: universe1
          value: .findmaxtest

     let -- universe1 = .findmaxtest

<sdvs.4.5.6> letsd
          name: loop.sd1
   state delta: u
        number: 1
     letsd -- loop.sd1 = u(1)

<sdvs.4.5.7> letsd
          name: loop.sd2
   state delta: u
        number: 2
     letsd -- loop.sd2 = u(2)

<sdvs.4.5.8> induct
    induction expression: .findmax.i
                    from: 2
                      to: .size + 1
       invariant list[]: formula(loop.sd1),formula(loop.sd2),
                         covering(.findmaxtest,universe1),
                         forall j (1 le j & j le .findmax.i - 1 ->
.x[j] le .m),
                         exists j ((1 le j & j le .size) & .x[j] = .m)
   comodification list[]: diff(universe1,union(m,findmax.i,findmaxtest\pc,findmaxtest))
```

```
modification list[]:  m,findmax.i,findmaxtest,findmaxtest\pc,diff(all,universe1)
       base proof[]:
       step proof[]:

induction -- .findmax.i from 2 to .size + 1

  open -- [sd pre: (true)
           comod: (all)
            post: (formula(loop.sd1),formula(loop.sd2),
                   covering(.findmaxtest,universe1),
                   forall j (1 le j & j le .findmax.i - 1
                             --> .x[j] le .m),
                   exists j ((1 le j & j le .size) & .x[j] = .m),
                   .findmax.i = 2)]
```

We now supply the proof for the base case of the induction. After we prove the two quantified statements, the base case will close automatically and the system will initiate the proof for the step case of the induction.

```
<sdvs.4.5.8.1.1> whynotgoal
  simplify?[no]:

 g(4) forall j (1 le j & j le 2 - 1 --> .x[j] le x\50)
 g(5) exists j ((1 le j & j le size\45) & .x[j] = x\50)

<sdvs.4.5.8.1.1> instantiate
      existential formula:  g
                   number:  5
   existential variable[]:  j
         instantiated by:  1
   existential variable[]:

        instantiate in goal 5 -- 1 for j.

<sdvs.4.5.8.1.2> notice
   term:  forall j(.x[1] le .m)

        notice -- forall j (.x[1] le .m)

<sdvs.4.5.8.1.3> usablequantifiers

 q(1) forall j (x\50 le x\50)

<sdvs.4.5.8.1.3> provebygeneralization
          prove universal formula:  g
                          number:  4
      number of universal formulas:  1
          using universal formula:  q
                          number:  1
        provebygeneralization -- forall j (1 le j & j le 2 - 1
                                           --> .x[j] le x\50)

      close -- 3 steps/applications

      open -- [sd pre: (.findmax.i ge 2,.findmax.i lt .size + 1,
                        formula(loop.sd1),formula(loop.sd2),
```

```
                covering(.findmaxtest,universe1),
                forall j (1 le j & j le .findmax.i - 1
                          --> .x[j] le .m),
                exists j ((1 le j & j le .size) & .x[j] = .m))
          comod: (diff(universe1,
                       union(m,findmax.i,findmaxtest\pc,findmaxtest)))
            mod: (m,findmax.i,findmaxtest,findmaxtest\pc,
                  diff(all,universe1))
           post: ([sd pre: (~(.findmax.i le .size))
                     comod: (all)
                       mod: (findmaxtest\pc)
                      post: (<adatr while i <= size
                                    loop
                                      max (i, ...);
                                      ...
                                    end loop;>)],
                  [sd pre: (.findmax.i le .size)
                     comod: (all)
                       mod: (findmaxtest\pc)
                      post: (<adatr while i <= size
                                    loop
                                      max (i, ...);
                                      ...
                                    end loop;>)],
                  covering(#findmaxtest,universe1),
                  forall j (1 le j & j le #findmax.i - 1
                            --> #x[j] le #m),
                  exists j ((1 le j & j le #size) & #x[j] = #m),
                  #findmax.i = .findmax.i + 1)]

Complete the proof.
```

We next advance to the point where the procedure call is made.

```
<sdvs.4.5.8.2.1> go
   until[]:  #findmaxtest\pc = at(findmaxtest.inner.max)

        apply -- [sd pre: (.findmax.i le .size)
                    comod: (all)
                      mod: (findmaxtest\pc)
                     post: (<adatr while i <= size
                                   loop
                                     max (i, ...);
                                     ...
                                   end loop;>)]

        instantiate in q(1) -- all top-level existential variables

        apply -- [sd pre: (true)
                    comod: (all)
                      mod: (findmaxtest\pc,findmaxtest)
                     post: (alldisjoint(findmaxtest,.findmaxtest,max.i,n),
                            covering(#findmaxtest,.findmaxtest,max.i,n),
                            declare(max.i,type(integer)),
                            declare(n,type(integer)),
                            <adatr max (i, ...)>)]
```

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (findmaxtest\pc,max.i,n)
           post: (#max.i = .findmax.i,#n = .m,
                  <adatr max (i, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (findmaxtest\pc)
           post: (#findmaxtest\pc = at(findmaxtest.inner.max),
                  <adatr max (i, ...)>)]

go -- breakpoint reached
```

We are now at the beginning of the call to procedure max; the call has been partially processed by declaring the formal variables of the procedure and binding to them the values of the actual arguments in the procedure call. (The last state delta applied simply issued the condition that informs the system that this is a place where the max.lemma might be applied.) If we continue to apply usable state deltas, symbolic execution will enter the body of max. Instead, we issue the *invokeadalemma* command.

```
<sdvs.4.5.8.2.6> invokeadalemma
   Ada lemma name: max.lemma

        invokeadalemma (no apply) -- [sd pre: (.findmaxtest\pc
                                                 = at(findmaxtest.inner.max),
                                               1 le .max.i,
                                               .max.i le .size,
                                               exists k ((1 le k &
                                                          k le .size) &
                                                         .x[k] = .n))
                                      comod: (all)
                                        mod: (findmaxtest\pc,n)
                                       post: (#n = .n or
                                                #n = .x[.max.i],
                                              #n ge .n,
                                              #n ge .x[.max.i],
                                              exists k ((1 le k &
                                                         k le .size) &
                                                        .x[k] = #n),
                                              #findmaxtest\pc
                                                 = exited(findmaxtest.inner.max),
                                              <adatr return;>)]

<sdvs.4.5.8.2.7> usable

 u(1) [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.max),1 le .max.i,
                .max.i le .size,
                exists k ((1 le k & k le .size) & .x[k] = .n))
       comod: (all)
         mod: (findmaxtest\pc,n)
        post: (#n = .n or #n = .x[.max.i],#n ge .n,#n ge .x[.max.i],
               exists k ((1 le k & k le .size) & .x[k] = #n),
               #findmaxtest\pc = exited(findmaxtest.inner.max),
```

```
              <adatr return;>)]

u(2) [sd pre: (~(.x[.max.i] gt .n),.max.i ge 1,.max.i le .size)
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr not (x (i) > n)>)]

u(3) [sd pre: (.x[.max.i] gt .n,.max.i ge 1,.max.i le .size)
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr if x (i) > n
                        then n := x (i);
                     end if;>)]


q(1) exists j ((1 le j & j le size\45) & .x[j] = m\62)

q(2) forall j (1 le j & j le findmax.i\61 - 1 --> .x[j] le m\62)
```

The `max.lemma` state delta was added to the usable list, but was not applied. Why not?

```
<sdvs.4.5.8.2.7> whynotapply
   state delta[ highest usable]: u
                         number: 1
 Because the following is not known to be true -- exists k ((1 le k &
                                                             k le .size) &
                                                            .x[k] = .n)
```

The problem is that the system does not see that the usable quantified formula `q(1)` is equivalent to the precondition of `u(1)`. In this case, it is because the two formulas use a different bound variable, but in general there could be many reasons. For the purpose of proving the needed precondition, we create, prove, and apply a statically true state delta.

```
<sdvs.4.5.8.2.7> prove
   state delta:
       [SD pre: true
       comod[]: all
         mod[]:
          post: exists k(1 le k & k le .size & .x[k] = .n)
       ]
       proof[]:

         open -- [sd pre: (true)
   comod: (all)
    post: (exists k ((1 le k & k le .size) &
       .x[k] = .n))]

 Complete the proof.

<sdvs.4.5.8.2.7.1> usablequantifiers

 q(1) exists j ((1 le j & j le size\45) & .x[j] = m\62)

 q(2) forall j (1 le j & j le findmax.i\61 - 1 --> .x[j] le m\62)
```

```
<sdvs.4.5.8.2.7.1> whynotgoal
   simplify?[no]:

 g(1) exists k ((1 le k & k le size\45) & .x[k] = m\62)

<sdvs.4.5.8.2.7.1> instantiate
       existential formula: q
                    number: 1
    existential variable[]: j
          instantiated by: j0
    existential variable[]:

          instantiate in q(1) -- j0 for j.

<sdvs.4.5.8.2.7.2> instantiate
       existential formula: g
                    number: 1
    existential variable[]: k
          instantiated by: j0
    existential variable[]:

          instantiate in goal 1 -- j0 for k.

         close -- 2 steps/applications

 Complete the proof.

<sdvs.4.5.8.2.8> usable

 u(1) [sd pre: (true)
       comod: (all)
        post: (exists k ((1 le k & k le .size) & .x[k] = .n))]

 u(2) [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.max),1 le .max.i,
                .max.i le .size,
                exists k ((1 le k & k le .size) & .x[k] = .n))
       comod: (all)
         mod: (findmaxtest\pc,n)
        post: (#n = .n or #n = .x[.max.i],#n ge .n,#n ge .x[.max.i],
               exists k ((1 le k & k le .size) & .x[k] = #n),
               #findmaxtest\pc = exited(findmaxtest.inner.max),
               <adatr return;>)]

 u(3) [sd pre: (~(.x[.max.i] gt .n),.max.i ge 1,.max.i le .size)
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr not (x (i) > n)>)]

 u(4) [sd pre: (.x[.max.i] gt .n,.max.i ge 1,.max.i le .size)
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr if x (i) > n
                          then n := x (i);
                      end if;>)]
```

```
q(1) exists j ((1 le j & j le size\45) & .x[j] = m\62)

q(2) forall j (1 le j & j le findmax.i\61 - 1 --> .x[j] le m\62)

<sdvs.4.5.8.2.8> whynotapply
   state delta[ highest usable]: u
                         number: 2
 Because the following is not known to be true -- exists k ((1 le k &
                                                              k le .size) &
                                                             .x[k] = .n)

<sdvs.4.5.8.2.8> apply
   sd/number[highest applicable/once]: u
                               number: 1
         apply -- [sd pre: (true)
                   comod: (all)
                    post: (exists k ((1 le k & k le .size) &
                                     .x[k] = .n))]

<sdvs.4.5.8.2.9> whynotapply
   state delta[ highest usable]: u
                         number: 2
 Quite applicable.
```

Having proved the statically true state delta, we applied it and found afterwards that u(2), which is the max.lemma, is now applicable.

```
<sdvs.4.5.8.2.9> apply
   sd/number[highest applicable/once]: u
                               number: 2
         apply -- [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.max),
                            1 le .max.i,.max.i le .size,
                            exists k ((1 le k & k le .size) &
                                      .x[k] = .n))
                   comod: (all)
                     mod: (findmaxtest\pc,n)
                    post: (#n = .n or #n = .x[.max.i],#n ge .n,
                           #n ge .x[.max.i],
                           exists k ((1 le k & k le .size) &
                                     .x[k] = #n),
                           #findmaxtest\pc = exited(findmaxtest.inner.max),
                           <adatr return;>)]

         non-trivial propagations -- n\75 = m\62 or n\75 = x\74

<sdvs.4.5.8.2.10> go
   until[]: #findmax.i = .findmax.i + 1

         apply -- [sd pre: (true)
                   comod: (all)
                     mod: (findmaxtest\pc)
                    post: (#findmaxtest\pc = exited(findmaxtest.inner.max),
                           <adatr max (i, ...)>)]

         apply -- [sd pre: (true)
                   comod: (all)
```

```
                mod: (findmaxtest\pc,m)
               post: (#m = .n,
                      <adatr max (i, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (findmaxtest\pc,findmaxtest,max.i,n)
           post: (covering(.findmaxtest,#findmaxtest,max.i,n),
                  undeclare(max.i,n),
                  <adatr max (i, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (findmaxtest\pc,findmax.i)
           post: (#findmax.i = .findmax.i + 1,
                  <adatr i := i + 1;>)]

go -- breakpoint reached
```

After the application of the `max.lemma`, the *go* command has advanced symbolic execution to the end of the `while` loop. Here, we have to prove the postconditions necessary to close the induction.

```
<sdvs.4.5.8.2.14>  usable

 u(1) [sd pre: (~(.findmax.i le .size))
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr while i <= size
                      loop
                        max (i, ...);
                        ...
                      end loop;>)]

 u(2) [sd pre: (.findmax.i le .size)
       comod: (all)
         mod: (findmaxtest\pc)
        post: (<adatr while i <= size
                      loop
                        max (i, ...);
                        ...
                      end loop;>)]


 q(1) exists k ((1 le k & k le size\45) & .x[k] = n\75)

 q(2) exists k ((1 le k & k le size\45) & .x[k] = m\62)

 q(3) exists j ((1 le j & j le size\45) & .x[j] = m\62)

 q(4) forall j (1 le j & j le findmax.i\61 - 1 --> .x[j] le m\62)

<sdvs.4.5.8.2.14>  whynotgoal
  simplify?[no]:

 g(4) forall j (1 le j & j le #findmax.i - 1 --> #x[j] le #m)
```

```
 g(5) exists j ((1 le j & j le #size) & #x[j] = #m)

<sdvs.4.5.8.2.14> ppeq
   expression: .m

 eqclass = n\75

<sdvs.4.5.8.2.14> instantiate
       existential formula: q
                    number: 1
    existential variable[]: k
          instantiated by: j1
    existential variable[]:

         instantiate in q(1) -- j1 for k.

         non-trivial propagations -- x\83 = m\62 or x\83 = x\74

<sdvs.4.5.8.2.15> instantiate
       existential formula: g
                    number: 5
    existential variable[]: j
          instantiated by: j1
    existential variable[]:

         instantiate in goal 5 -- j1 for j.

         non-trivial propagations -- x\83 = m\62 or x\83 = x\74
```

That takes care of the existential goal. Now what of the universal one?

```
<sdvs.4.5.8.2.16> whynotgoal
   simplify?[no]:

 g(4) forall j (1 le j & j le #findmax.i - 1 --> #x[j] le #m)

<sdvs.4.5.8.2.16> usablequantifiers

 q(1) exists k ((1 le k & k le size\45) & .x[k] = n\75)

 q(2) exists k ((1 le k & k le size\45) & .x[k] = m\62)

 q(3) exists j ((1 le j & j le size\45) & .x[j] = m\62)

 q(4) forall j (1 le j & j le findmax.i\61 - 1 --> .x[j] le m\62)

<sdvs.4.5.8.2.16> ppeq
   expression: .findmax.i

 eqclass = 1 + findmax.i\61
           findmax.i\61 + 1
```

This shows that q(4) will not suffice for g(4); it only covers the values of j through .findmax.i - 2, whereas we need values through .findmax.i - 1. So we create another

universal formula to deal with the case `j .findmax.i - 1`, and then use a *provebygeneralization* command to prove `g(4)`.

```
<sdvs.4.5.8.2.16> notice
   term: forall j(.x[.findmax.i-1] le .m)

         notice - 'forall j (.x[.findmax.i - 1] le .m)

         non-trivial propagations -- x\83 = m\62 or x\83 = x\74

<sdvs.4.5.8.2.17> usablequantifiers

 q(1) forall j (x\74 le n\75)

 q(2) exists k ((1 le k & k le size\45) & .x[k] = n\75)

 q(3) exists k ((1 le k & k le size\45) & .x[k] = m\62)

 q(4) exists j ((1 le j & j le size\45) & .x[j] = m\62)

 q(5) forall j (1 le j & j le findmax.i\61 - 1 --> .x[j] le m\62)

<sdvs.4.5.8.2.17> provebygeneralization
            prove universal formula: g
                             number: 4
       number of universal formulas: 2
            using universal formula: q
                             number: 1
            using universal formula: q
                             number: 5
          provebygeneralization -- forall j (1 le j &
                                             j  le (1 + findmax.i\61) - 1
                                             --> .x[j] le n\75)

          non-trivial propagations -- x\83 = m\62 or x\83 = x\74

        close -- 17 steps/applications

      join induction cases -- [sd pre: (2 le .size + 1)
                               comod: (all,
                                       diff(universe1,
                                            union(m,findmax.i,
                                                  findmaxtest\pc,
                                                  findmaxtest)))
                                mod: (m,findmax.i,findmaxtest,
                                      findmaxtest\pc,diff(all,universe1))
                               post: (#findmax.i = .size + 1,
                                      formula(loop.sd1),
                                      formula(loop.sd2),
                                      covering(#findmaxtest,universe1),
                                      forall j (1 le j &
                                                j le #findmax.i - 1
                                                --> #x[j] le #m),
                                      exists j ((1 le j & j le #size) &
                                                #x[j] = #m))]

  Complete the proof.
```

```
<sdvs.4.5.9> go
   until[]:  #findmaxtest\pc = exited(findmaxtest.inner.findmax)

     apply -- [sd pre: (~(.findmax.i le .size))
               comod: (all)
                 mod: (findmaxtest\pc)
                post: (<adatr while i <= size
                                loop
                                  max (i, ...);
                                  ...
                                end loop;>)]

     apply -- [sd pre: (true)
               comod: (all)
                 mod: (findmaxtest\pc,findmaxtest,findmax.i)
                post: (covering(.findmaxtest,#findmaxtest,findmax.i),
                       undeclare(findmax.i),
                       <adatr i : integer>)]

     apply -- [sd pre: (true)
               comod: (all)
                 mod: (findmaxtest\pc)
                post: (#findmaxtest\pc = exited(findmaxtest.inner.findmax),
                       <adatr null;>)]

     go -- breakpoint reached
```

Once the goals were proved, the induction closed automatically; the subsequent *go* command has brought us to the end of `findmax`. Now we only have to prove the postconditions of `findmax.lemma` itself. This is handled easily by the *instantiation* and *provebygeneralization* commands.

```
<sdvs.4.5.12> whynotgoal
   simplify?[no]:

 g(1) forall j (1 le j & j le size\45 --> .x[j] le #m)
 g(2) exists j ((1 le j & j le size\45) & #m = .x[j])

<sdvs.4.5.12> usablequantifiers

 q(1) exists j ((1 le j & j le size\45) & .x[j] = m\87)

 q(2) forall j (1 le j & j le (1 + size\45) - 1 --> .x[j] le m\87)

<sdvs.4.5.12> provebygeneralization
            prove universal formula:  g
                             number:  1
       number of universal formulas:  1
            using universal formula:  q
                             number:  2
     provebygeneralization -- forall j (1 le j & j le size\45
                                        --> .x[j] le m\87)

<sdvs.4.5.13> usablequantifiers
```

```
q(1) forall j (1 le j & j le size\45 --> .x[j] le m\87)

q(2) exists j ((1 le j & j le size\45) & .x[j] = m\87)

q(3) forall j (1 le j & j le (1 + size\45) - 1 --> .x[j] le m\87)

<sdvs.4.5.13> instantiate
      existential formula: q
                   number: 2
   existential variable[]: j
         instantiated by: j2
   existential variable[]:

    instantiate in q(2) -- j2 for j.

<sdvs.4.5.14> instantiate
      existential formula: g
                   number: 2
   existential variable[]: j
         instantiated by: j2
   existential variable[]:

    instantiate in goal 2 -- j2 for j.

  close -- 14 steps/applications

 close -- 5 steps/applications

 proveadalemma -- [sd pre: (.findmaxtest\pc = at(findmaxtest.inner.findmax),
                            1 le .size)
                   comod: (all)
                     mod: (findmaxtest\pc,m)
                    post: (forall j (1 le j & j le .size --> .x[j] le #m),
                           exists j ((1 le j & j le .size) &
                                     #m = .x[j]),
                           #findmaxtest\pc
                              = exited(findmaxtest.inner.findmax))]

<sdvs.4>
```

The proof of `findmax.lemma` is complete. The saved form of it follows:

```
; -*- Syntax: Common-lisp; Package: USER; Mode: LISP -*-~%

(defproof findmax.proof
   "(adatr \"testproofs/findmax.a\",
     createadalemma max.lemma
                file: \"testproofs/findmax.a\"
           procedure: max
       qualified name: findmaxtest.inner.max
         precondition: (1 le .max.i,.max.i le .size,
exists k ((1 le k & k le .size) & .x[k] = .n))
            mod list: (n)
        postcondition: (#n = .n or #n = .x[.max.i],#n ge .n,#n ge .x[.max.i],
exists k ((1 le k & k le .size) & .x[k] = #n)),
```

```
    proveadalemma max.lemma
      proof:
cases .x[.max.i] gt .n
  then proof:
    (go #findmaxtest\\pc = exited(findmaxtest.inner.max),
     instantiate (k=.max.i) in g(4))
  else proof: go #findmaxtest\\pc = exited(findmaxtest.inner.max),
    createadalemma findmax.lemma
                file: \"testproofs/findmax.a\"
           procedure: findmax
      qualified name: findmaxtest.inner.findmax
        precondition: (1 le .size)
            mod list: (m)
       postcondition: (forall j (1 le j & j le .size --> .x[j] le #m),
exists j ((1 le j & j le .size) & #m = .x[j])),
    proveadalemma findmax.lemma
      proof:
 (applydecls,
  go .findmax.i = 2,
  let universe1 = .findmaxtest,
  letsd loop.sd1 = u(1),
  letsd loop.sd2 = u(2),
  induct on:     .findmax.i
    from:        2
    to:          .size + 1
    invariants:  (formula(loop.sd1),formula(loop.sd2),
  covering(.findmaxtest,universe1),
  forall j (1 le j & j le .findmax.i - 1 --> .x[j] le .m),
  exists j ((1 le j & j le .size) & .x[j] = .m))
    comodlist:   (diff(universe1,
       union(m,findmax.i,findmaxtest\\pc,findmaxtest)))
    modlist:     (m,findmax.i,findmaxtest,findmaxtest\\pc,
  diff(all,universe1))
    base proof:
      (instantiate (j-1) in g(5),
       notice forall j (.x[1] le .m),
       provebygeneralization g(4)
 using: (q(1)))
    step proof:
      (go #findmaxtest\\pc = at(findmaxtest.inner.max),
       invokeadalemma max.lemma,
       prove [sd pre: (true)
       comod: (all)
post: (exists k ((1 le k & k le .size) &
   .x[k] = .n))]
 proof:
   (instantiate (j=j0) in q(1),
    instantiate (k=j0) in g(1)),
       apply u(1),
       apply u(2),
       go #findmax.i = .findmax.i + 1,
       instantiate (k=j1) in q(1),
       instantiate (j=j1) in g(5),
       notice forall j (.x[.findmax.i - 1] le .m),
       provebygeneralization g(4)
 using: (q( ),q(5))),
  go #fi  axtest\\pc = exited(findmaxtest.inner.findmax),
```

```
provebygeneralization g(1)
  using: (q(2)),
instantiate (j=j2) in q(2),
instantiate (j=j2) in g(2)))")
```

# 5 Conclusions

This prototype offline characterization mechanism has been shown to be useful in eliminating some of the steps formerly required of proofs involving Ada procedure calls. Through the examples presented here, it is shown to work effectively in some simple contexts. Its eventual usefulness is not entirely apparent, as only a few examples have been performed, and those examples do not involve complex procedures that are called at more than one place. The utility of this feature should become more clear when we scale up to prove the correctness of large Ada programs, where characterization of subroutines will play a much larger role in proof organization and efficiency. Another important consideration is the modularization of the proofs—as programs get larger, the corresponding proofs get longer and more difficult to follow. The offline characterization mechanism should do much to ameliorate this difficulty.

# References


[1] L. Marcus, "SDVS 9 Users' Manual," Technical Report ATR-90(5778)-4, The Aerospace Corporation, September 1990.

[2] D. F. Martin, "A Formal Description of the Incremental Translation of Stage 2 Ada into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-90(5778)-5, The Aerospace Corporation, September 1990.

[3] J. E. Doner and J. V. Cook, "Offline Characterization of Procedures in the State Delta Verification System (SDVS)," Technical Report ATR-90(8590)-5, The Aerospace Corporation, September 1990.